SOFTWARE TESTING AND AUTOMATION

MODULE-01

INTRODUCTION

- **What is Testing?**
- Testing is the process of evaluating a system or its component(s) with the intent to find whether it satisfies the specified requirements or not.
- According to ANSI/IEEE 1059 standard, Testing can be defined as - A process of analysing a software item to detect the differences between existing and required conditions (that is defects/errors/bugs) and to evaluate the features of the software item

- **Who does Testing?**
- It depends on the process and the associated stakeholders of the project(s).
- In most cases, the following professionals are involved in testing a system within their respective capacities –

  - Software Tester
  - Software Developer
  - Project Lead/Manager
  - End User

- **When to Start Testing?**
- During the requirement gathering phase, the analysis and verification of requirements are also considered as testing.
- Reviewing the design in the design phase with the intent to improve the design is also considered as testing.
- Testing performed by a developer on completion of the code is also categorized as testing.

- **When to Stop Testing?**
- Testing Deadlines
- Completion of test case execution
- Completion of functional and code coverage to a certain point
- Bug rate falls below a certain level and no high-priority bugs are identified
- Management decision

TEST CASE

- A TEST CASE is a set of conditions or variables under which a tester will determine whether a system under test satisfies requirements or works correctly.

Structure of a Test Case

- **Test Case ID** – Unique number (e.g., TC_01).
- **Test Scenario** – What feature you are testing.
- **Preconditions** – Things that must be true before test (e.g., app is installed).
- **Test Steps** – Actions the tester performs.
- **Test Data** – Inputs provided.
- **Expected Result** – What should happen.
- **Actual Result** – What actually happened (filled after execution).
- **Status** – Pass/Fail.

EXAMPLE

TEST CASE FOR LOGIN PAGE

- **Test Case ID:** TC_01
- **Test Scenario:** Verify login with valid credentials.
- **Precondition:** User must have a registered account.
- **Test Steps:**
  - Open the login page.
  - Enter **Username = "john123"**.
  - Enter **Password = "Test@123"**.
  - Click on **Login** button.
- **Test Data:** Username → "john123", Password → "Test@123".
- **Expected Result:** User should be redirected to the home/dashboard page.
- **Actual Result:** (Tester fills after running).

Status: Pass/Fail.

Entry Criteria for Different Levels of Testing

- In software testing, before starting any testing phase, **certain conditions must be met**.
  These conditions are called **Entry Criteria**.
- It ensures:
- The testing team has everything they need.
- The phase starts only when the system is ready.
- No time is wasted testing something incomplete.

1. Unit Testing (Testing individual modules/functions)

✅ **Business requirements are present** – So we know what the software should do.

✅ **Functional requirements & specifications available** – Details about each feature are clear.

✅ **High-level design document available** – The architecture or design of the system is defined.

✅ **Planning phase completed** – Testing approach and scope are already planned.

- 👉 *Meaning:* Before testing a single "unit" of code, we must know the requirements, design, and plan.

2. Integration Testing (Testing combined modules)

✅ **All modules are unit tested and available** – Each module works fine individually.

✅ **Code is completed** – No missing parts.

✅ **All priority bugs fixed and closed** – Critical errors found earlier must be resolved.

✅ **Integration test plans, scenarios, and cases are ready** – Clear steps for combining modules are written.

👉 *Meaning:* Before integration testing, individual modules must be tested and stable.

3. System Testing (Testing the complete system)

✅ **Integration of modules completed** – The system is assembled and works together.

✅ **Passed exit criteria of Integration Testing** – Integration testing was successful.

✅ **All priority bugs fixed and closed** – No major issues remaining.

✅ **System testing environment set up** – Hardware/software environment is ready.

✅ **Test cases/scripts ready** – Detailed test steps are prepared.

👉 *Meaning:* Before system testing, the full system should be built, stable, and ready in a proper environment.

4. Acceptance Testing (Final testing by end-users/business)

✅ **System Testing exit criteria met** – System testing is successfully completed.

✅ **Business requirements met** – The product works as per business goals.

✅ **Acceptance test environment set up** – Test environment ready for end-users.

✅ **Test cases/scripts ready** – Steps for acceptance testing are documented.

👉 *Meaning:* Before acceptance testing, the system must already be tested and aligned with business needs.

WHAT IS SOFTWARE TESTING

- Software testing is a method for finding out if the software meets requirements and is error-free.
- Software testing is the process of determining if a piece of software is accurate by taking into account all of its characteristics (reliability, scalability, portability, Reusability and usability) and analyzing how its various components operate in order to detect any bugs, faults or flaws.

- **What are the Benefits of Software Testing?**
- **Cost Effective** : If flaws are found sooner in the software testing process, fixing them is less expensive.
- **Security:** It assists in eradicating hazards and issues early.
- **Product quality**: Any software product must meet these criteria. Testing guarantees that buyers get a high-quality product.
- **Customer satisfaction:** Providing consumers with contentment is the primary goal of every product.

- **Definition of Software testing can be considered into:**

1) Process: Testing is a process rather than a single activity.

2) All Life Cycle Activities: Software Development Life Cycle (SDLC).

- The process of designing tests early in the life cycle can help to prevent defects from beings Introduced in the code. It is referred as test design .
- The test basis includes documents such as the requirements and design specifications.

3) Static Testing: It can test and find defects without executing code. Static Testing is done during verification process. This testing includes reviewing of the documents (including source code) and static analysis. This is useful and cost effective way of testing. For example: reviewing, walkthrough, inspection, etc.

4) Dynamic Testing: In dynamic testing the software code is executed to demonstrate the result unit testing, integration testing, system testing, etc.

- 5) **Planning**: We need to plan as what we want to do. We control the test activities, we report on testing progress and the status of the software under test.
- 6) **Preparation:** We need to choose what testing we will do, by selecting test conditions and designing test cases.
- 7) **Evaluation:** During evaluation we must check the results and evaluate the software under test and the completion criteria, which helps us to decide whether we have finished testing and whether the software product has passed the tests.
- 8) **Software products and related work products:** Along with the testing of code the testing of requirement and design specifications and also the related documents like operation, user and training material is equally important.

What is Verification?

- Verification is a static practice of verifying documents, design, code and program.
- It includes all the activities associated with producing high quality software: inspection, design analysis and specification analysis. It is a relatively objective process.
- Methods of Verification:

✓ Static Testing

✓ Walkthrough

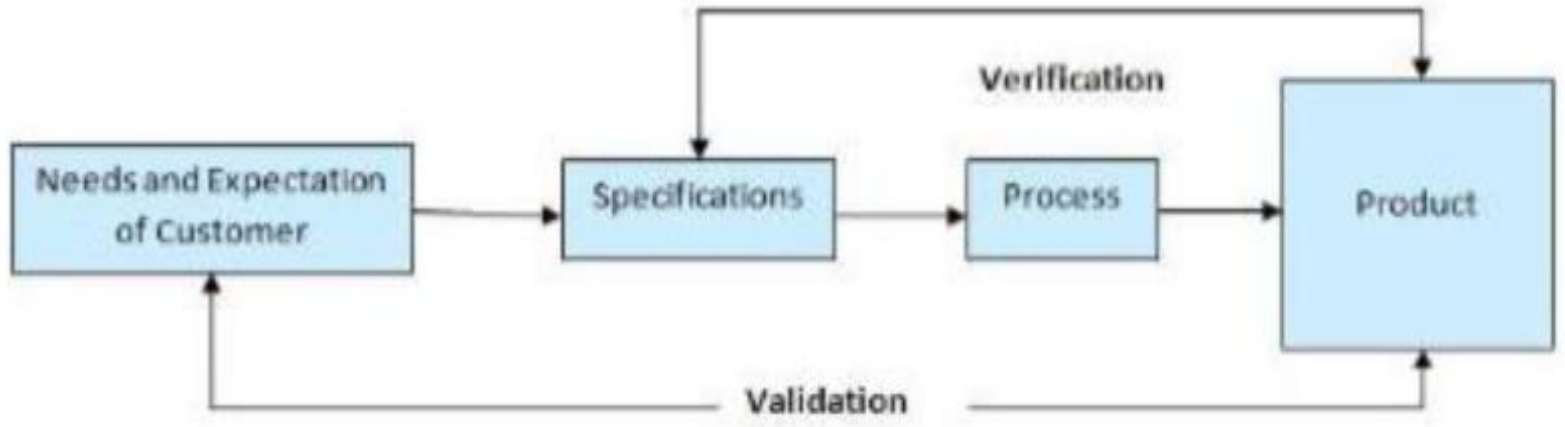
✓ Inspection

✓ Review.

What is Validation?

- Validation is the process of evaluating the final product to check whether the software meets the customer expectations and requirements.
- It is a dynamic mechanism of validating and testing the actual product.
- Methods of Validation:

✓ Dynamic Testing



Software verification and validation

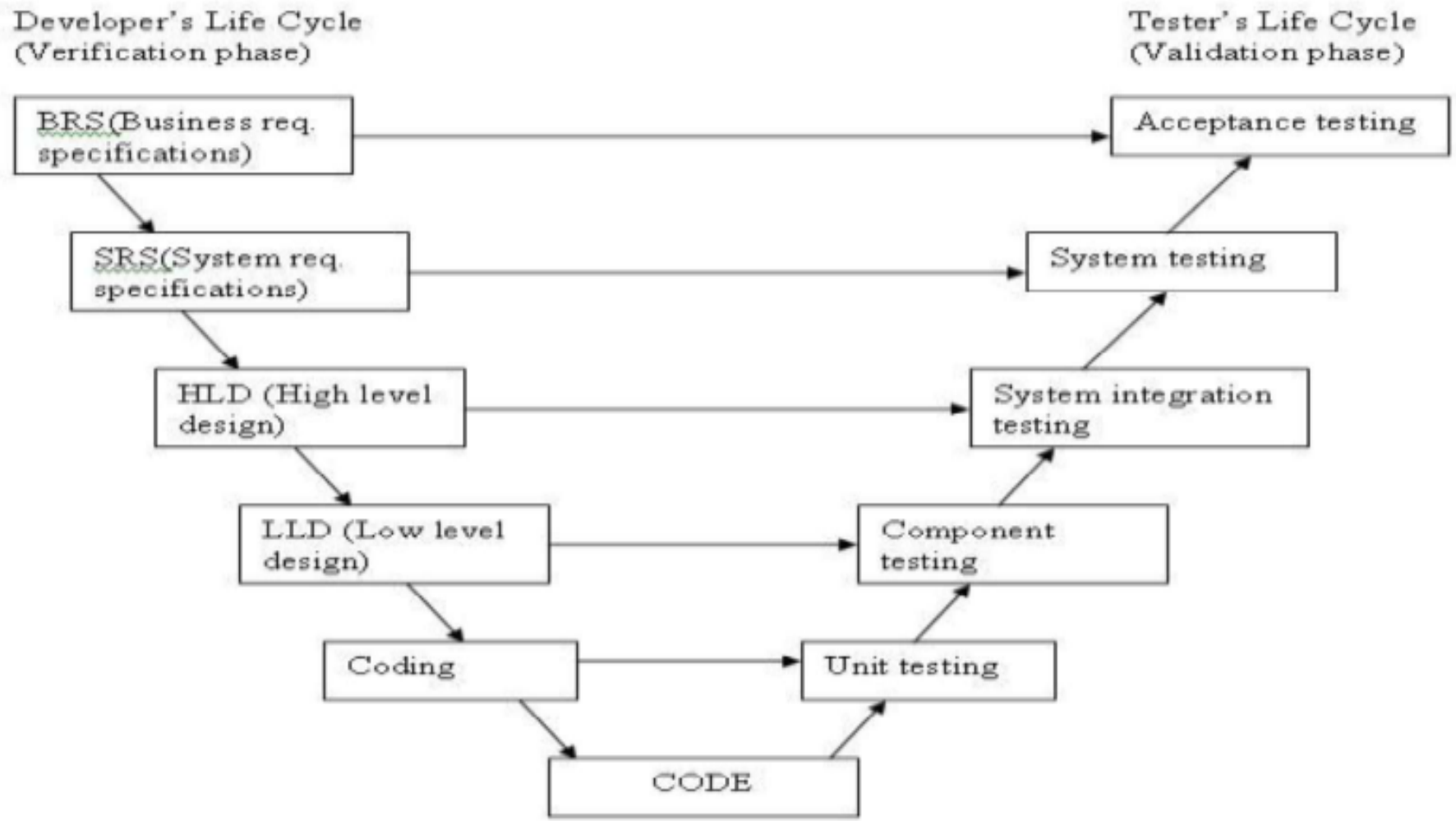
Differences between Verification & Validation:

Verification	Validation
1. Verification is a static practice of verifying documents, design, code and program.	1. Validation is a dynamic mechanism of validating and testing the actual product.
2. It does not involve executing the code.	2. It always involves executing the code.
3. It is human based checking of documents and files.	3. It is computer based execution of program.
4. Verification uses methods like inspections, reviews, walkthroughs, and Desk-checking etc.	4. Validation uses methods like black box (functional) testing, gray box testing, and white box (structural) testing etc.
5. Verification is to check whether the software conforms to specifications.	5. Validation is to check whether software meets the customer expectations and requirements.
6. It can catch errors that validation cannot catch. It is low level exercise.	6. It can catch errors that verification cannot catch. It is High Level Exercise.
7. Target is requirements specification, application and software architecture, high level, complete design, and database design etc.	7. Target is actual product-a unit, a module, a bent of integrated modules, and effective final product.
8. Verification is done by QA team to ensure that the software is as per the specifications in the SRS document.	8. Validation is carried out with the involvement of testing team.
9. It generally comes first-done before validation.	9. It generally follows after verification.

V-MODEL OF SOFTWARE TESTING

- The V-Model provides a systematic and visual representation of the software development process.
- V Model also referred to as the Verification and Validation Model.
- Testing of the device is planned in parallel with a corresponding stage of development.

Diagram of V-model:



- **The various phases of the V-model are as follows:**
- **Requirements** like BRS and SRS begin the life cycle model just like the waterfall model. But, in this model before development is started, a system test plan is created. The test plan focuses on meeting the functionality specified in the requirements gathering.
- **The high-level design (HLD)** phase focuses on system architecture and design. It provides overview of solution, platform, system, product and service/process. An integration test plan is created in this phase as well in order to test the pieces of the software systems ability to work together.
- **The low-level design (LLD)** phase is where the actual software components are designed. It defines the actual logic for each and every component of the system. Class diagram with all the methods and relation between classes comes under LLD. Component tests are created in this phase as well.
- **The implementation** where all coding takes place. Once coding is complete, the path of execution phase is, again, continues up the right side of the V where the test plans developed earlier are now put to use.
- **Coding:** This is at the bottom of the V-Shape model. Module design is converted into code by developers. Unit Testing is performed by the developers on the code written by them.

Advantages of V-model:

- Testing activities like planning, test designing happens well before coding. This saves a lot of time. Hence higher chance of success over the waterfall model.
- Proactive defect tracking that is defects are found at early stage.
- Avoids the downward flow of the defects.
- Works well for small projects where requirements are easily understood.
- Simple and easy to use.

Disadvantages of V-model:

- Very rigid and least flexible.
- Software is developed during the implementation phase, so no early prototypes of the software are produced.
- If any changes happen in midway, then the test documents along with requirement documents has to be updated.

When to use the V-model:

- The V-shaped model should be used for small to medium sized projects where requirements are clearly defined and fixed.
- The V-Shaped model should be chosen when ample technical resources are available with needed technical expertise.
- High confidence of customer is required for choosing the V-Shaped model approach.
- Since, no prototypes are produced, there is a very high risk involved in meeting customer expectations.

Taxonomy of Bugs:

- A defect is an error or a bug, in the application which is created.
- A programmer while designing and building the software can make mistakes or error. These mistakes or errors mean that there are flaws in the software. These are called defects.
- When actual result deviates from the expected result while testing a software application or product then it results into a defect.
- Hence, any deviation from the specification mentioned in the differently like bug, issue, incidents or problem.

- **This Defect report or Bug report consists of the following information:**
- **Defect ID:** Every bug or defect has its unique identification number
- **Defect Description:** This includes the abstract of the issue.
- **Product Version:** This includes the product version of the application in which the defect is found.
- **Detail Steps:** This includes the detailed steps of the issue with the screenshots attached so that developers can recreate it.
- **Date Raised:** This includes the Date when the bug is reported.
- **Reported By:** This includes the details of the tester who reported the bug like Name and ID.

- **Status:** This field includes the Status of the defect like New, Assigned, open, Retest, Verification, Closed, Failed, Deferred, etc.
- **Fixed by:** This field includes the details of the developer who fixed it like Name and ID.
- **Date Closed:** This includes the Date when the bug is closed Severity Based on the severity (Critical, Major or Minor) it tells us about impact of the defect or bug in the software application.
- **Priority:** Based on the Priority set (High/Medium/Low) the order of fixing the defect can be made.

The Importance of a Bug:

- A reasonable metric for bug importance is:
- **Importance = frequency * (correction_cost + installation_cost + consequential_cost)**
- Frequency: How often does the bug occur?
- Correction Cost: What does it cost to discover and correct a bug?
- Installation Cost: Depends on the number of installations. Cost is small for single user systems, large for widely used systems (e.g., Windows XP)
- Consequences: Depends on what kind of awards are made to the victims of the bug.

Type of Software Testing

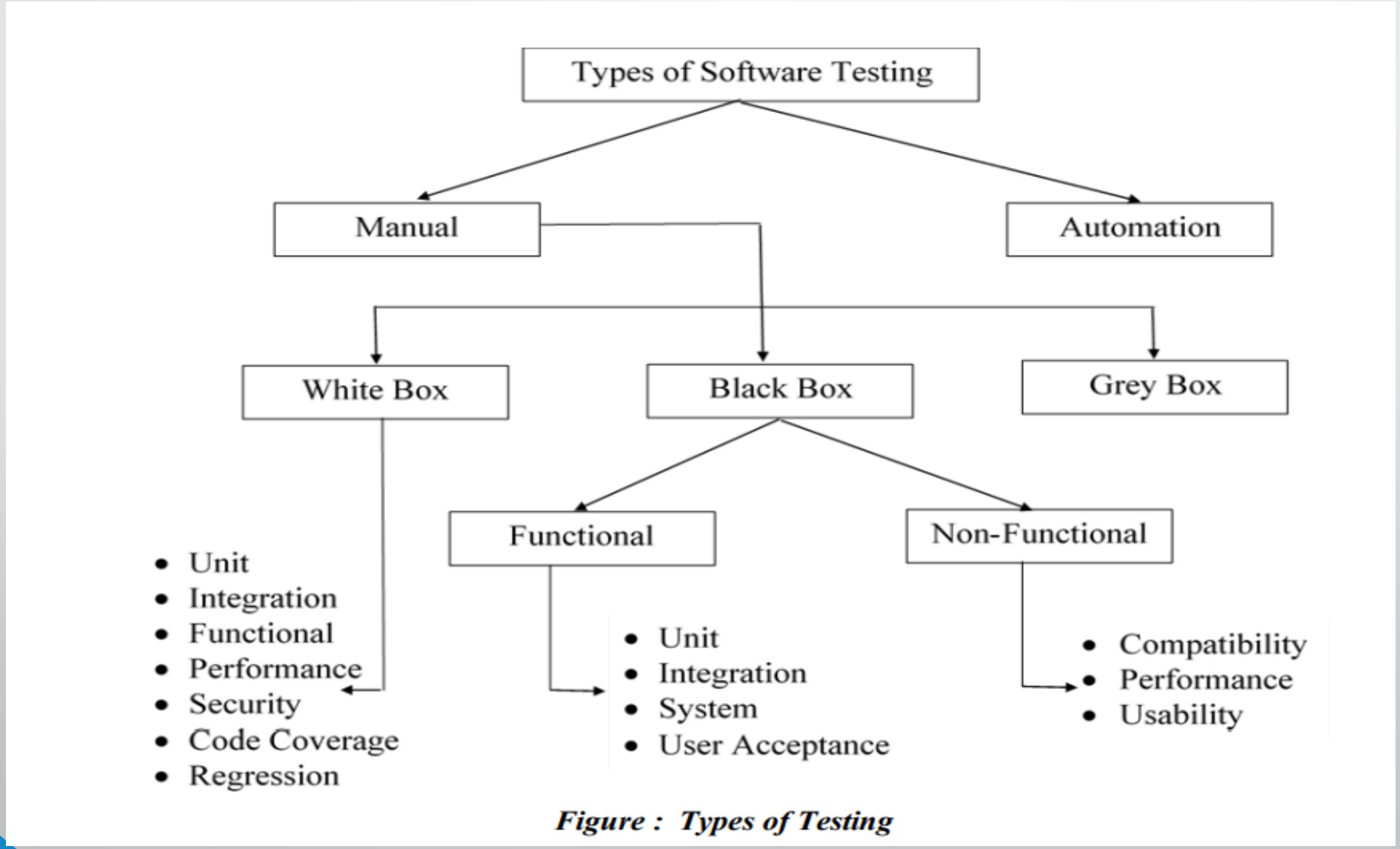


Figure : Types of Testing

- 1. **Manual testing:** The process of checking the functionality of an application as per the customer needs without taking any help of automation tools is known as manual testing.
- While performing the manual testing on any application, we do not need any specific knowledge of any testing tool, rather than have a proper understanding of the product so we can easily prepare the test document.
- Manual testing can be further divided into three types of testing, which are as follows:
- **White box testing Black box testing Grey box testing.**

- 2. **Automation testing:** Automation testing is a process of converting any manual test cases into the test scripts with the help of automation tools or any programming language.
- With the help of automation testing, we can enhance the speed of our test execution because here, we do not require any human efforts.

Manual Testing Vs Automation Testing

Manual Testing	Automation Testing
In manual testing, the test cases are executed by the human tester.	In automated testing, the test cases are executed by the software tools.
Manual testing is time-consuming.	Automation testing is faster than manual testing.
Manual testing takes up human resources.	Automation testing takes up automation tools and trained employees.
Exploratory testing is possible in manual testing.	Exploratory testing is not possible in automation testing.
Initial Investment is less	Initial Investment is more

White Box Testing (Glass Box / Clear Box Testing)

- White Box Testing is a **software testing technique** where the **internal structure, code, and logic of the program** are tested.
- In this testing, the tester **knows the code**, understands the **flow of inputs and outputs**, and checks whether the **code is working as intended**.
- It is mainly done by **developers** or **testers with programming knowledge**.

- **Why is it called "White Box"?**
- Imagine the software is inside a transparent box.
- You can see inside the box (the code and logic) and test it.
- This is the opposite of Black Box Testing, where you only test inputs/outputs without knowing the code inside.
- **Objectives of White Box Testing**
- Verify the flow of inputs and outputs in the application.
- Ensure all decision branches and conditions work correctly.
- Find hidden errors, such as:
  - Loops that never end
  - Unreachable code
  - Security vulnerabilities
- Optimize the code for better performance.

Techniques of White Box Testing

There are several structured ways to test the internal logic:

1. Statement Coverage

- Every line of code (statement) should be executed at least once.
  - ✔ Ensures no line of code is left untested.

2. Branch Coverage

- Every **decision (true/false)** should be tested.
  - ✔ Example: `if (x > 0) { ... } else { ... }` → both conditions tested.

3. Path Coverage

- Test **all possible paths** through the program.
  - ✔ Example: If a function has 2 decisions, there may be 4 paths to test.

4. Loop Testing

- Ensures that **for, while, do-while loops** are tested:
  - Zero times
  - Once
  - Multiple times

5. Condition Coverage

- Each **Boolean expression** in the code should evaluate both **true** and **false**.

Example of White Box Testing

Let's say we have a simple code:

```c
int checkNumber(int x) {
    if (x > 0)
        return 1;   // Positive
    else if (x < 0)
        return -1;  // Negative
    else
        return 0;   // Zero
}
```

Test Cases for White Box:

1. Input = 10 → Covers condition `x > 0` → returns 1
2. Input = -5 → Covers condition `x < 0` → returns -1
3. Input = 0 → Covers condition `x == 0` → returns 0

Black Box Testing (Behavioral / Functional Testing)

- Black Box Testing is a **software testing method** where the tester examines the **functionality of the software without knowing the internal code, logic, or structure**.
- The tester only provides **inputs** and checks the **outputs**.
- Focus is on **"What the system does"** rather than **"How it does it."**
- Mainly done by **QA testers** who may not need programming knowledge.

- **Why is it called "Black Box"?**
- Imagine the software is inside a sealed black box.
- You cannot see the internal code or logic.
- You only interact with it through inputs and outputs.
- **Objectives of Black Box Testing**
- Verify whether the software meets the business requirements.
- Validate the functionality, usability, and performance of the system.
- Detect incorrect outputs, missing functions, and interface issues.
- Ensure the system works properly from a user’s perspective.

- **Techniques of Black Box Testing**

There are several structured approaches to designing black box test cases:

1. Equivalence Partitioning

Input values are divided into **groups (partitions)**, and one value from each group is tested.
✓ Example: For input range **1–100**, test 10 (valid), -5 (invalid), 150 (invalid).

2. Boundary Value Analysis (BVA)

Errors are most common at **boundaries** of input values.
✓ Example: For input range **1–100**, test 0, 1, 100, 101.

3. Decision Table Testing

Used when there are **multiple input conditions** and **different outputs**.
✓ Example: Login form with conditions (Valid Username, Valid Password).

State Transition Testing

Tests how the system behaves when it **changes from one state to another**.
✓ Example: ATM – from "Card Inserted" → "PIN Entered" → "Transaction".

5. Error Guessing

Based on tester's **experience and intuition**.
✓ Example: Entering special characters in a name field.

- **Example of Black Box Testing**
- Suppose we have a **Login Page**:
- Input: Username, Password
- Output: Success Message / Error Message
- **Test Cases:**
- Valid username + valid password → Login successful.
- Valid username + invalid password → Error message.
- Invalid username + valid password → Error message.
- Blank fields → Error message.
- Special characters in username → Error message.
- ☞ Notice: The tester **didn't check the code**, only checked if the software works correctly with different inputs.

Feature	White Box Testing	Black Box Testing
Knowledge Needed	Code knowledge required	No code knowledge needed
Focus	Internal logic, structure	Functionality, behavior
Tester	Developers, Testers with coding skills	QA Testers, End-users
Main Techniques	Path coverage, loop testing, condition coverage	Equivalence partitioning, BVA, decision tables
Used in	Unit testing, security testing	System, acceptance, regression testing

Grey Box Testing (Hybrid Testing)

- **Grey Box Testing** is a **software testing method** that combines aspects of both **Black Box Testing** and **White Box Testing**.
- Tester has **partial knowledge of the internal structure/code**.
- Focus is on **both functionality (like Black Box)** and **internal logic (like White Box)**.
- The goal is to design more effective test cases by using some code knowledge while still testing from a **user perspective**.

- **Why is it called "Grey Box"?**
- In Black Box Testing, the internal code is completely hidden.
- In White Box Testing, the tester sees the full internal code.
- In Grey Box Testing, the tester has limited visibility (not full code, but partial details like design docs, database schemas, or APIs).
  ☞ That's why it's in-between → Grey (mix of Black & White).
- **Objectives of Grey Box Testing**
- To identify defects due to improper usage of code, data flow, or system behavior.
- To ensure both functional correctness and internal system efficiency.
- To find security loopholes, data handling issues, and integration bugs.
- To test end-to-end workflows with some awareness of internal logic.

- **Techniques of Grey Box Testing**

1. Matrix Testing

Focuses on testing relationships between **functions, modules, and data**.

Helps verify whether requirements and test cases are fully mapped.

2. Regression Testing with Code Knowledge

Knowing the **changed code area**, testers focus on impacted features.

3. Pattern Testing

Uses past experiences and known defect patterns to test new systems.

4. Orthogonal Array Testing

A mathematical approach to test combinations of inputs while reducing test cases.

5. Database Testing

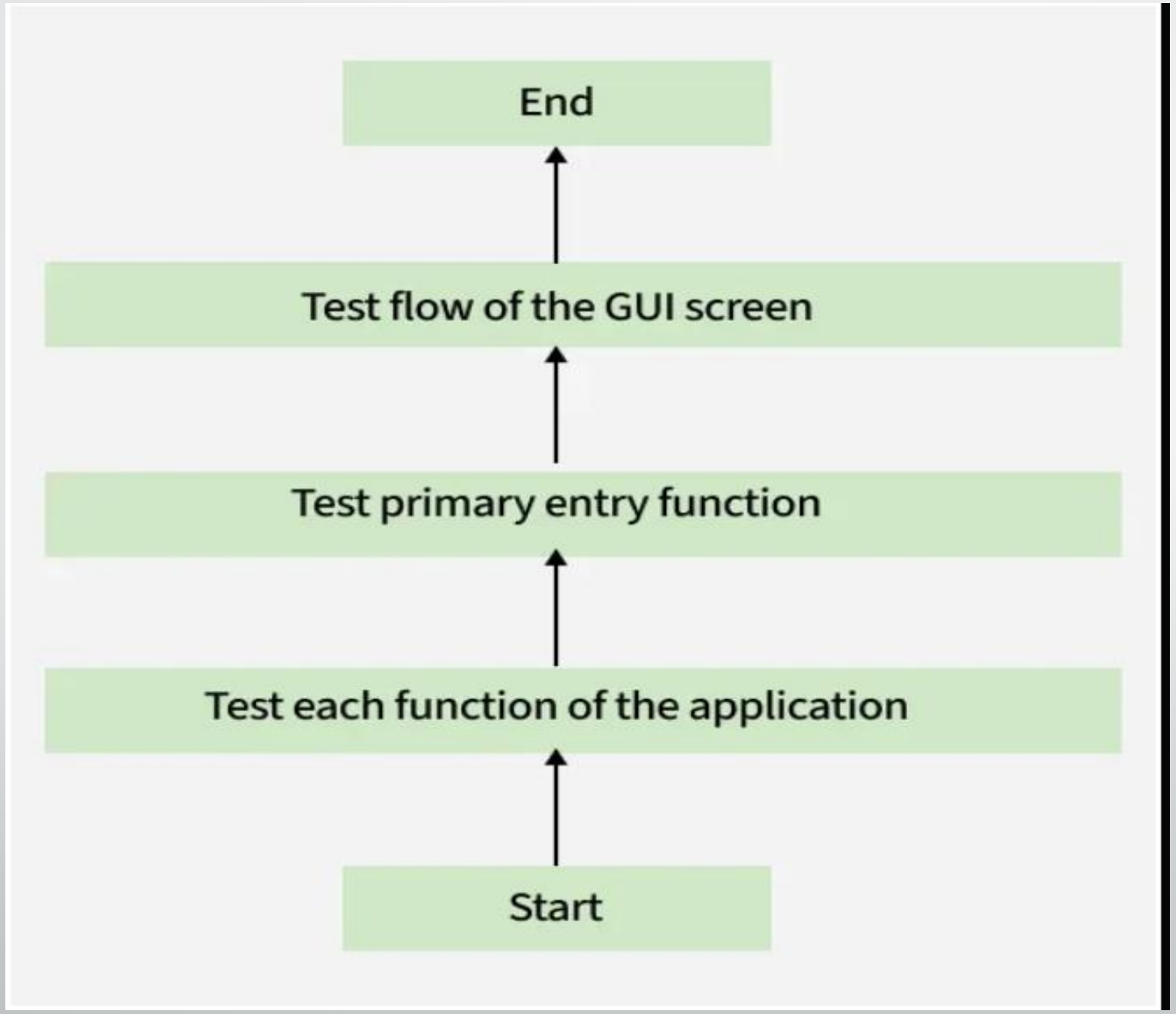
Testers check **queries, stored procedures, triggers**, using limited DB knowledge.

- **Example of Grey Box Testing**
- Example : E-commerce Website
- Black Box Way: Test "Add to Cart" → Check if items are added correctly.
- White Box Way: Check internal functions, SQL queries, memory handling.
- Grey Box Way: Tester knows the database schema → Tests edge cases like:
  - What happens if cart item ID doesn't exist?
  - Does system allow duplicate entries in DB?
  - Does checkout correctly update multiple tables?

Functional Testing - Software Testing

- Functional Testing is defined as a Type of Software Testing that verifies that each function of the Software Application works in conformance with the requirements and specifications.
- This testing is not concerned with the source code of the application.
- Each functionality of the software application is tested by providing appropriate test input, expecting the output, and comparing the actual output with the expected output.
- This testing focuses on checking the user interface, APIs, Database, Security, Client or Server Application, and functionality of the Application Under Test.
- Functional testing can be performed manually or through automation, depending on the needs of the project.

- **Purpose of Functional Testing**
- Functional testing mainly involves black box testing and can be done manually or using automation. The purpose of functional testing is to:
- **Test each function of the application:** Functional testing tests each function of the application by providing the appropriate input and verifying the output against the functional requirements of the application.
- **Test primary entry function:** In functional testing, the tester tests each entry function of the application to check all the entry and exit points.
- **Test flow of the GUI screen:** In functional testing, the flow of the GUI screen is checked so that the user can navigate throughout the application.

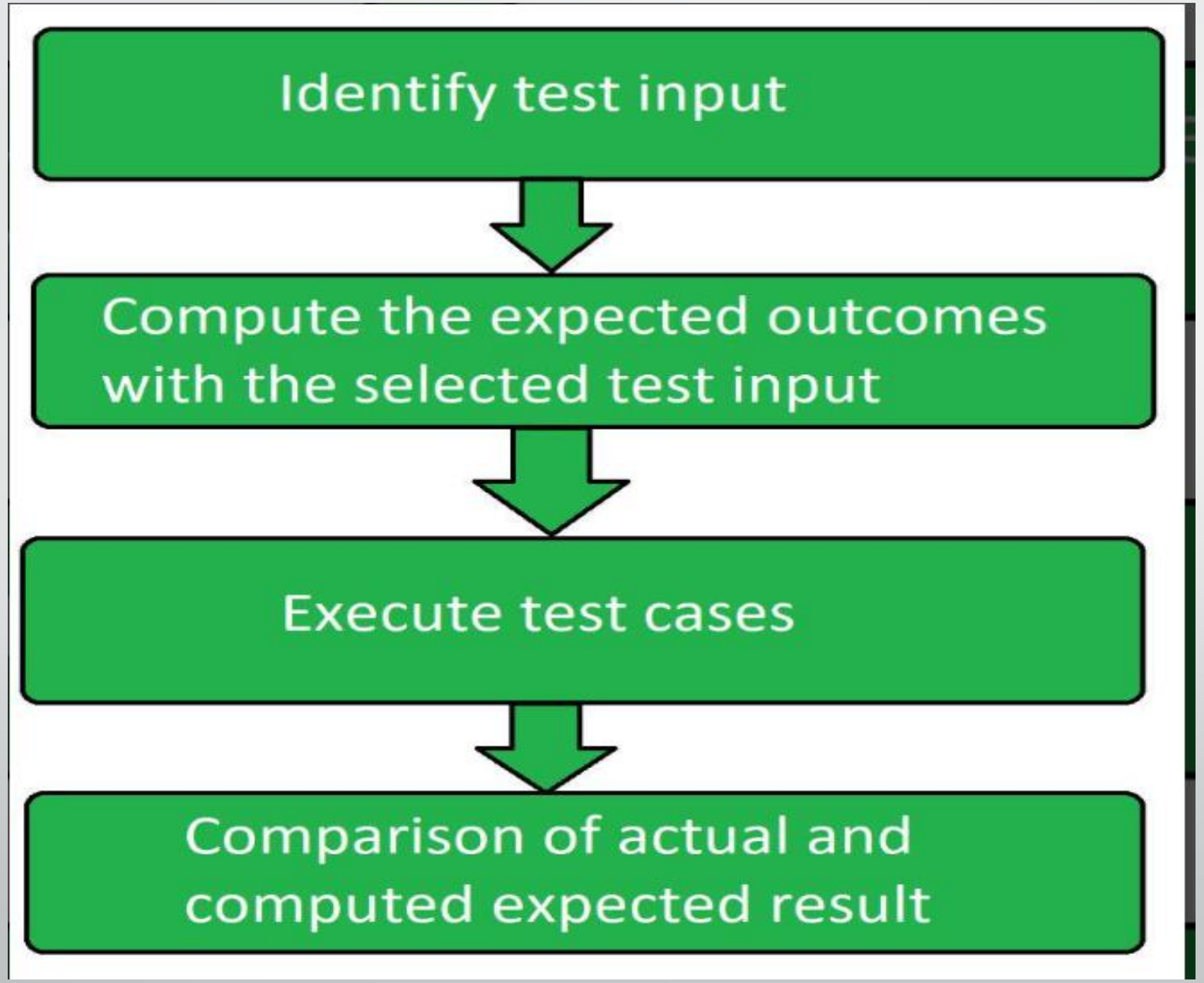

End
Test flow of the GUI screen
Test primary entry function
Test each function of the application
Start

- **What to Test in Functional Testing?**
- The goal of functional testing is to make sure the app's features work as they should. It focuses on these key areas:
- **Basic Usability**: This checks if users can easily navigate the app without any trouble. It's all about making sure the experience is smooth.
- **Main Functions**: Functional testing also looks at the app is a core features to verify they are working correctly, just as they're meant to.
- **Accessibility**: This ensures the app is accessible to everyone, including users with disabilities. It checks whether accessibility features are in place and functioning properly.
- **Error Handling**: Lastly, it tests how the app handles errors. Are the right error messages shown when something goes wrong? This part verify users are informed when an issue arises.

- **Functional Testing Process**
- Functional testing follows four steps to check if your app or software works as expected:

- **Step 1. Identify test input**
- The first step in functional testing is to identify the functionality that needs to be tested. This involves determining which core features of the application, such as login, registration, or payment processing, should be tested. It also includes testing usability functions like buttons, forms, and navigation links.
- Additionally, error conditions must be considered, such as invalid inputs or scenarios where the system fails.
- **Step 2. Compute expected outcomes**
- Once the test inputs are identified, the next step is to calculate the expected outcomes based on the functional specifications of the system. Here, you define what should happen for each test case based on the input data. For instance, when entering a valid username and password, the expected output could be that the user is successfully redirected to the homepage.
- On the other hand, for invalid login attempts (like an incorrect password), the expected outcome might be an error message prompting the user to try again.

- **Step 3. Execute test cases**
- After defining the expected outcomes, the next step is to execute the test cases using the identified inputs. This is where you run the tests and see how the system behaves. Test cases can be executed manually or using automation tools such as Selenium, JUnit, or TestNG.
- **Step 4. Compare the actual and expected output**
- In the final step of functional testing, you compare the actual output of the system to the expected output. If the system's response matches what was expected, then the functionality is confirmed to be working correctly. This step ensures that the system functions as expected, providing the correct results under various conditions.


Identify test input
Compute the expected outcomes with the selected test input
Execute test cases
Comparison of actual and computed expected result

- **Type of Functional Testing Techniques**
- There are various types of functional Testing which are as follows:
- **Unit testing:** It is the type of functional testing technique where the individual units or modules of the application are tested. It ensures that each module is working correctly.
- **Integration testing:** Combined individual units are tested as a group and expose the faults in the interaction between the integrated units.
- **System Testing:** It is a type of testing that is performed on a completely integrated system to evaluate the compliance of the system with the corresponding requirements. In system testing, integration testing passed components are taken as input.
- **User Acceptance Testing (UAT):** User Acceptance Testing (UAT) serves the purpose of ensuring that the software meets the business requirements and is ready for deployment by validating its functionality in a real-world environment. It allows end-users to test the software to ensure it meets their needs and operates as expected, helping to identify and fix any issues before the final release.

Testing techniques

- Typical black-box test design techniques include:
- **Decision table testing**
- **Boundary value analysis**
- **Equivalence partitioning**

DECISION TABLE-BASED TESTING

- Decision Table Based Testing is a **systematic black-box testing technique** used to test the behavior of a system when there are **multiple input conditions** and **corresponding actions (outputs)**.
  It is particularly useful when the system's behavior depends on **combinations of inputs** rather than individual inputs.
- It works like a **truth table** (from logic), where all possible combinations of conditions are mapped with the corresponding actions. Because of this, decision tables are often called **Cause–Effect Tables**:
- **Causes** → Input conditions.
- **Effects** → Output actions.

- **Why Decision Table Testing is Needed?**
- When a system involves **complex business rules** with multiple input combinations.
- Helps in identifying **uncovered scenarios** that might not be obvious in normal test case design.
- Prevents **redundant test cases**, since we only choose relevant combinations.
- Provides a **clear visual representation** of conditions vs. actions.
- **Example use cases:**
- Insurance premium calculation (based on age, medical history, coverage type).
- Loan eligibility (based on income, credit score, collateral).
- Flight ticket pricing (based on passenger type, seat class, travel season).

- **How to Use decision tables for test designing?**
- The first task is to identify a suitable function or subsystem which reacts according to a combination of inputs or events.
- The system should not contain too many inputs otherwise the number of combinations will become unmanageable.
- It is better to deal with large numbers of conditions by dividing them into subsets and dealing with the subsets one at a time.
- Once you have identified the aspects that need to be combined, then you put them into a table listing all the combinations for True and False for each of the aspects.

Check whether given value for a equilateral, isosceles , Scalene triangle or can't from a triangle

Input data decision Table

RULES		R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11
Conditions	C1: a < b + c	F	T	T	T	T	T	T	T	T	T	T
	C2 : b < a + c	-	F	T	T	T	T	T	T	T	T	T
	C3 : c < a + b	-	-	F	T	T	T	T	T	T	T	T
	C4 : a = b	-	-	-	T	T	T	T	F	F	F	F
	C5 : a = c	-	-	-	T	T	F	F	T	T	F	F
	C6 : b = c	-	-	-	T	F	T	F	T	F	T	F
	a1 : Not a triangle	X	X	X								
Actions	a2 : Scalene triangle											X
	a3 : Isosceles triangle							X		X	X	
	a4 : Equilateral triangle				X							
	a5 : Impossible					X	X		X			

Triangle Problem –Decision Table Test cases for input data

Case Id	Description	Input Data			Expected Output	Actual Output	Status	Comments
		a	b	C				
1	Enter the value of a, b and c Such that a is not less than sum of two sides	20	5	5	Message should be displayed can't form a triangle			
2	Enter the value of a, b and c Such that b is not less than sum of two sides and a is less than sum of other two sides	3	15	11	Message should be displayed can't form a triangle			
3	Enter the value of a, b and c Such that c is not less than sum of two sides and a and b is less than sum of other two Sides	4	5	20	Message should be displayed can't form a triangle			
4	Enter the value a, b and c satisfying precondition and a=b, b=c and c=a	5	5	5	Should display the message Equilateral triangle			
5	Enter the value a ,b and c satisfying precondition and a=b and b ≠ c	10	10	9	Should display the message Isosceles triangle			
6	Enter the value a, b and c satisfying precondition and a ≠b , b ≠ c and c ≠ a	5	6	7	Should display the message Scalene triangle			

Decision Base Table for Login Screen

The condition is simple if the user provides correct username and password the user will be redirected to the homepage. If any of the input is wrong, an error message will be displayed.

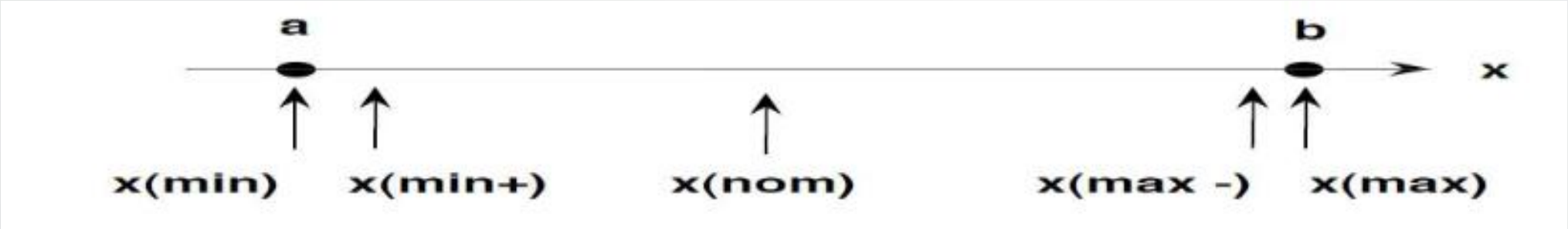
Conditions	Rule 1	Rule 2	Rule 3	Rule 4
Username (T/F)	F	T	F	T
Password (T/F)	F	F	T	T
Output (E/H)	E	E	E	H

Legend:
T – Correct username/password
F – Wrong username/password
E – Error message is displayed
H – Home screen is displayed

Interpretation:
Case 1 – Username and password both were wrong. The user is shown an error message.
Case 2 – Username was correct, but the password was wrong. The user is shown an error message.
Case 3 – Username was wrong, but the password was correct. The user is shown an error message.
Case 4 – Username and password both were correct, and the user navigated to homepage

BOUNDARY VALUE TESTING

- Boundary testing is the process of testing between extreme ends or boundaries between partitions of the input values.
- So these extreme ends like Start- End, Lower- Upper, Maximum-Minimum, Just Inside-Just-Outside values are called boundary values and the testing is called "boundary testing".
- The basic idea in boundary value testing is to select input variable values at their:
- Minimum
- Just above the minimum
- A nominal value
- Just below the maximum
- Maximum



- There are two independent considerations that apply to input domain testing.
- The first asks whether or not we are concerned with invalid values of variables.
- Normal boundary value testing is concerned only with valid values of the input variables. Robust boundary value testing considers invalid and valid variable values.
- The second consideration is whether we make the "single fault" assumption common to reliability theory.
- This assumes that faults are due to incorrect values of a single variable. Taken together, the two considerations yield four variations of boundary value testing:
- Normal boundary value testing
- Robust boundary value testing
- Worst-case boundary value testing
- Robust worst-case boundary value testing

- **Normal Boundary Value Testing**
- **Definition:** Test only the boundary values **inside the valid domain** (no invalids).
- For an input range [min, max], the test cases are:
  - **min, min+1, max–1, max**
- Total = **4n + 1 test cases** for *n* variables (since one "nominal"/typical value is also tested).
- **Example:** Input range 1–10.
  ☞ Test cases = {1, 2, 9, 10, (and one mid value e.g., 5)}
- **Graph (x-axis = input values, y-axis = test case selection):**

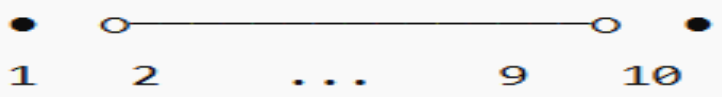
1. Normal Boundary Value Testing

- **Definition:** Test only the boundary values **inside the valid domain** (no invalids).
- For an input range [min, max], the test cases are:
  - **min, min+1, max–1, max**
- Total = **4n + 1 test cases** for *n* variables (since one "nominal"/typical value is also tested).

Example: Input range 1–10.

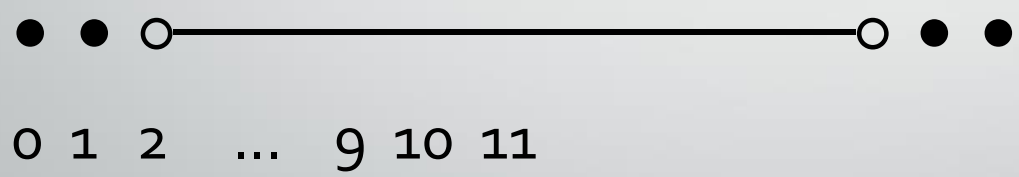
👉 Test cases = {1, 2, 9, 10, (and one mid value e.g., 5)}

Graph (x-axis = input values, y-axis = test case selection):



- Solid dots (●) = chosen boundary values.
- Open dots (○) = ignored middle values.

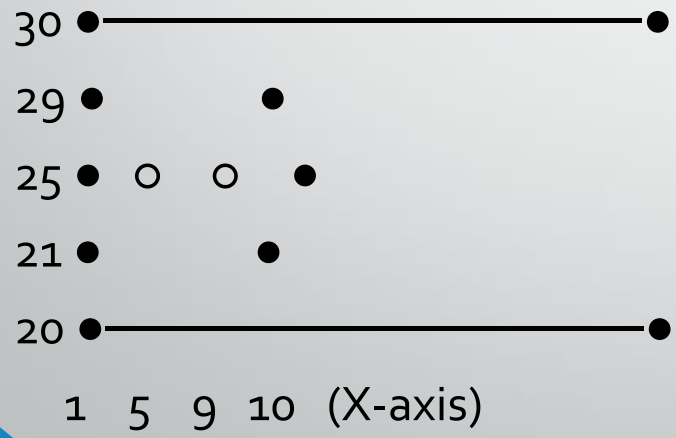
- **2. Robust Boundary Value Testing**
- **Definition:** Extends Normal BVA by also including **just outside the boundary values** (invalid inputs).
- For input range [min, max]:
  - **min–1, min, min+1, max–1, max, max+1**
- Total = **6n test cases** for *n* variables.
- **Example:** Input range 1–10.
  ☞ Test cases = {0, 1, 2, 9, 10, 11}
- **Graph:**



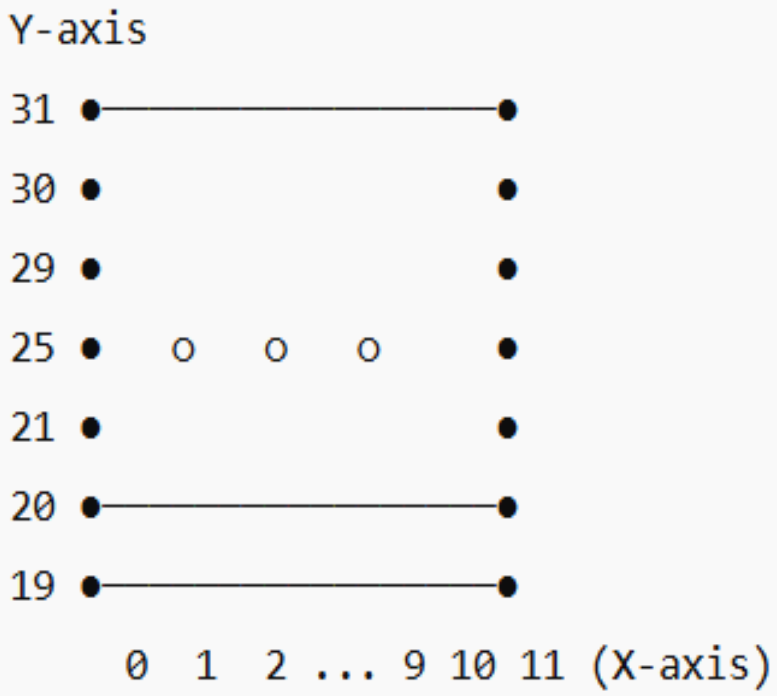
0 1 2 … 9 10 11

Now it checks **outside values** (0, 11) along with valid boundaries.

- **Worst-case Boundary Value Testing**
- **Definition:** Considers **all combinations** of boundary values for **multiple input variables**.
- Instead of testing each variable separately, we take **Cartesian product** of boundary values.
- Total = **(4n + 1)^n** test cases.
- **Example:** Suppose two variables:
- X range: 1–10 ⟶ {1, 2, 9, 10, 5}
- Y range: 20–30 ⟶ {20, 21, 29, 30, 25}
- ☞ All combinations ⟶ 5 × 5 = **25 test cases**
- **Graph (2D grid for X vs Y):**



- **Robust Worst-case Boundary Value Testing**
- **Definition:** The most exhaustive approach — includes **all combinations of normal + invalid (outside) boundaries** for multiple variables.
- For each variable: {min–1, min, min+1, max–1, max, max+1}.
- Total = **(6n)^n test cases** (huge in number).
- **Example:** Two variables:
- X range: 1–10 ⟶ {0, 1, 2, 9, 10, 11}
- Y range: 20–30 ⟶ {19, 20, 21, 29, 30, 31}
- ☞ All combinations ⟶ 6 × 6 = **36 test cases**
- **Graph (2D grid for X vs Y):**



- Covers both **valid** and **invalid** combinations at all boundaries.

Technique	Values Tested per Variable	Includes Invalids?	Test Case Growth
Normal BVA	min, min+1, max–1, max (+ mid)	✗ No	4n + 1
Robust BVA	min–1, min, min+1, max–1, max, max+1	✓ Yes	6n
Worst-case BVA	All combinations of Normal BVA values	✗ No	(4n+1)^n
Robust Worst-case BVA	All combinations of Robust BVA values	✓ Yes	(6n)^n

EQUIVALENCE CLASS TESTING

- **. Introduction**
- Equivalence Class Testing (ECT) is a **black-box testing technique** that reduces the number of test cases by dividing input data into **partitions (classes)** where the system is expected to behave the same way.
- 👉 **Idea:** Instead of testing every possible input, test **one representative value** from each class (partition). If one value in a class works, others are assumed to work too.

- **Why Equivalence Class Testing?**
- Input domains are often large or infinite (e.g., all integers from 1–10,000).
- Testing every value is impossible.
- ECT helps:
  - Minimize test cases without losing coverage.
  - Identify valid and invalid inputs.
  - Systematically cover inputs without redundancy.
- **Principle of Equivalence Partitioning**
- Identify the input domain (range of valid values).
- Divide the domain into equivalence classes:
  - Valid classes → inputs system should accept.
  - Invalid classes → inputs system should reject.
- From each class, pick one representative test case.

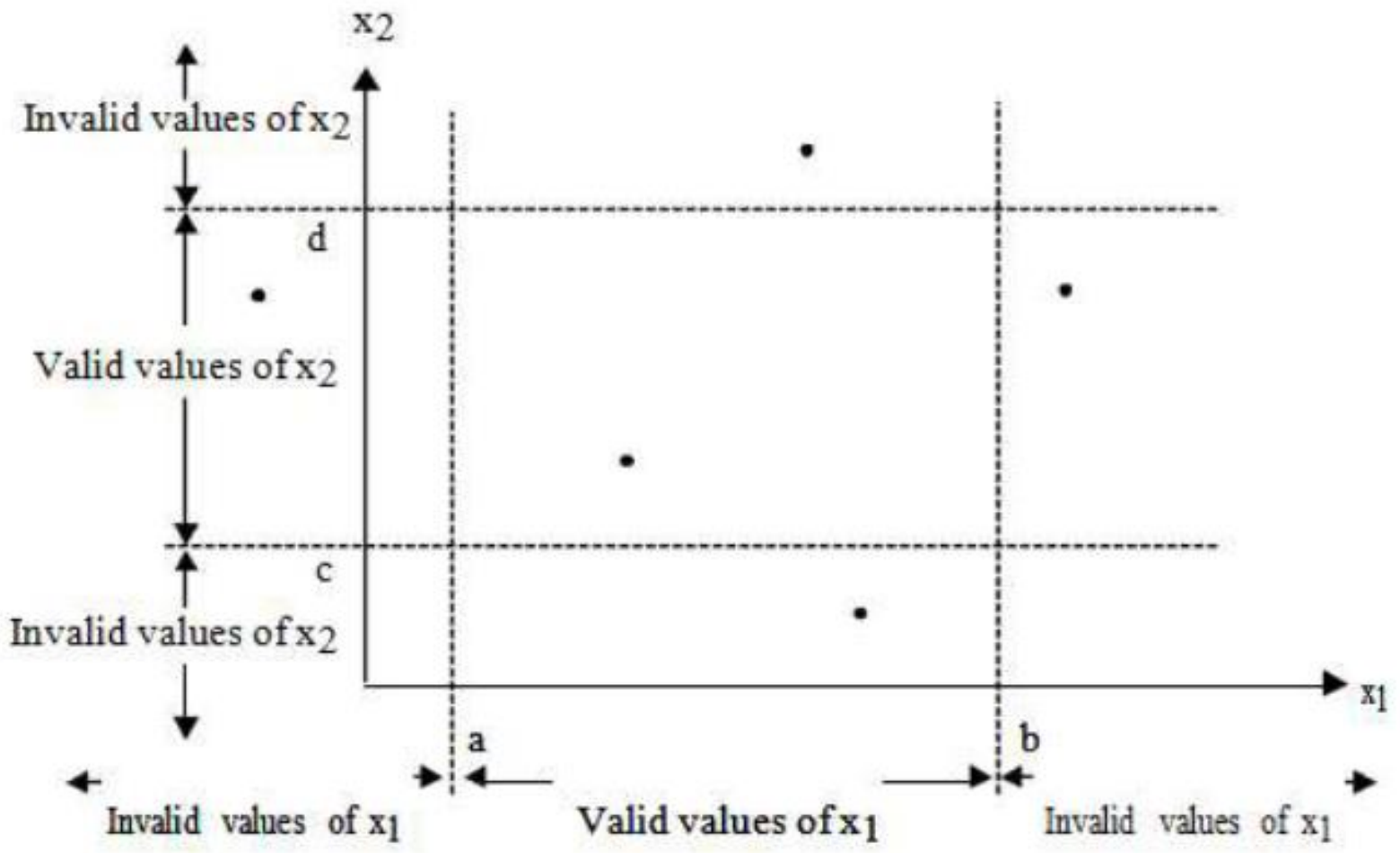
- **Example**
- **Requirement:** A field accepts age between **18 and 60** (inclusive).
- ☞ **Input domain:** All integers (… 0, 1, 2, …, 100, …).
- We divide into equivalence classes:
- **Valid class:** [18 – 60] ⟶ any value between 18 and 60.
- **Invalid class 1:** Age < 18 ⟶ e.g., 10.
- **Invalid class 2:** Age > 60 ⟶ e.g., 75.
- **Test Cases (one from each class):**
- TC1: 25 (Valid class) ⟶ Accepted.
- TC2: 10 (Invalid class 1) ⟶ Rejected.
- TC3: 75 (Invalid class 2) ⟶ Rejected.
- Instead of testing all values (0–100), only **3 test cases** are enough.

Traditional Equivalence Class Testing

Traditional equivalence class testing echoes the process of boundary value testing. Figure shows test cases for a function F of two variables x_1 and x_2. The extension to more realistic cases of n variables proceeds as follows:

1. Test F for valid values of all variables.
2. If step 1 is successful, then test F for invalid values of x_1 with valid values of the remaining variables. Any failure will be due to a problem with an invalid value of x_1.
3. Repeat step 2 for the remaining variables.

One clear advantage of this process is that it focuses on finding faults due to invalid data.



Traditional equivalence class test cases.